Санкт-Петербургский государственный университет
Прикладная математика и информатика
Исследование операций и принятие решений в задачах оптимизации, управления и экономики

Голякова Алена Андреевна

Вычисление и оценки линейной сложности бинарных последовательностей

Выпускная квалификационная работа

Научный руководитель:
к. ф.-м. н., доцент И. В. Агафонова

Рецензент:
к. ф.-м. н., доцент Н. С. Григорьева

Санкт-Петербург
2017

Saint Petersburg State University
Applied Mathematics and Computer Science
Operation Research and Decision Making in Optimisation, Control and Economics Problems

Golyakova Alyona Andreevna

Linear complexity of binary sequences: calculation and evaluation

Graduation Project

Scientific Supervisor:
PhD, Associate professor
I. V. Agafonova

Reviewer:
PhD, Associate Professor
N. S. Grigoryeva

Saint Petersburg
2017

Оглавление

Введение

В поточном шифре бинарное сообщение шифруется путем добавления поэлементно по модулю 2 псевдослучайной последовательности битов, в результате чего битовая строка предположительно трудно расшифровывается и, следовательно, может передаваться по открытым каналам связи. Необходимо, чтобы последовательности, полученные таким образом, были криптографически сильными в том смысле, что их восстановление по известному фрагменту последовательных символов было либо невозможно, либо требовало больших ресурсов.

Известно, что такое свойство последовательности, как линейная сложность, значимо в определении криптографически сильных последовательностей.

Данная работа посвящена исследованию задачи нахождения k-error линейной сложности бинарной последовательности и методов ее решения. При этом написаны программы нахождения точного решения k-error линейной сложности для следующих частных случаев:

1. $k = 0$ (Berlekamp–Massey Algorithm, [1]);
2. $k = 2^n$, где n — целое число (Stamp,Martin, [2]).

Основной целью работы является применение к рассматриваемой задаче одного из приближенных алгоритмов (а именно — генетического) и нахождение приближенного решения задачи при оптимальных значениях параметров.

Кроме того, проводится сравнение операторов генетического алгоритма, исследуется зависимость полученных результатов от выбранных операторов и параметров метода. Алгоритм реализован на языке Python 3.5.

Глава 1

Линейная сложность

1.1. Определения и примеры

Дана бесконечная последовательность $s = s_0, s_1, \ldots$ (либо конечная последовательность $s = s_0, s_1, \ldots, s_{t-1}$) с элементами из поля K.

Определение 1.1.1. *Говорят, что s является линейной рекуррентной последовательностью порядка L $(L > 0)$, если для членов этой последовательности выполняется соотношение*

$$s_j = \sum_{k=0}^{L-1} c_k s_{L-k} \tag{1.1}$$

для любых $j = L, L+1, \ldots$ (или для любых $j = L, L+1, \ldots, t-1$ соответственно), где $c_0, c_1, \ldots, c_{L-1} \in \{0, 1\}$.

Это уравнение называют *линейным рекуррентным отношением порядка L*. По данному соотношению вводится полином $C(X)$ следующим образом:

$$C(X) = X^L + c_{L-1}X^{L-1} + \ldots + c_1 X + c_0. \tag{1.2}$$

Он называется *характеристическим полиномом*, соответствующим параметрам $c_0, c_1, \ldots, c_{L-1}$.

Определение 1.1.2. *Минимальное L, такое что s является линейной рекуррентной последовательностью порядка L, называется линейной сложностью последовательности s. Для нулевой последовательности s значение L принимается равным нулю.*

Линейную сложность последовательности s будем обозначать $L(s)$.

Рекуррентное отношение минимального порядка называют *минимальным рекуррентным отношением*, а характеристический полином минимальной степени — *минимальным характеристическим полиномом*.

Как уже отмечалось ранее, линейная сложность является одной из основных характеристик, отвечающих за безопасность потоковых шифров. Для ее нахождения были найдены эффективные алгоритмы, но только для случаев бинарных последовательностей с периодом 2^n. Один из таких алгоритмов был предложен Xiao G. (см [3]).

Также существует эффективный метод нахождения линейной сложности конечной или периодической последовательности, описанный в источнике [1], известный как алгоритм Берлекэмпа–Мэсси (Berlekamp–Massey Algorithm, BMA). Код алгоритма находится в приложении А.

BMA требует только $2n$ последовательных битов для полного определения последовательности с линейной сложностью n.

Отсюда получаем, что перехвативший $2n$ последовательных элементов последовательности s (где n — линейная сложность) может эффективно восстановить всю псевдослучайную последовательность, и, следовательно, в случае поточного шифра все засекреченное сообщение может быть восстановлено.

Пример 1.1.1. Рассмотрим 20-периодичную последовательность s с циклом

$$s^{20} = 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0.$$

Ее линейная сложность, посчитанная при помощи алгоритма Берлекэмпа-Мэсси, равна 19.

Если считать s конечной последовательностью длины 20, то ее линей-

ная сложность будет равна 10.

Таким образом, криптографически сильная последовательность должна иметь высокую линейную сложность. В своей работе [4] Rueppel доказал это формально. При этом большая линейная сложность является необходимым, но не достаточным условием для криптостойкости. Например, последовательность

$$s = \underbrace{(0, 0, \ldots, 1)}_{n} \tag{1.3}$$

имеет линейную сложность $L = n$ (максимально возможная), но она криптографически слабая, так как исходное сообщение будет оставлено практически в неизменном виде. Поэтому требуются дополнительные измерения для определения криптографически более сильной последовательности. В этом отношении становится полезен профиль линейной сложности.

Пусть дана бесконечная двоичная последовательность $s = s_0, s_1, \ldots$ Через L_N обозначим линейную сложность подпоследовательности $s^N = s_0, s_1, \ldots, s_{N-1}$, $N \geqslant 0$.

Последовательность $L_1, L_2, \ldots$ называется *профилем линейной сложности последовательности* s.

В случае конечной последовательности профиль линейной сложности состоит из $L_1, L_2, \ldots, L_n$.

Рассмотрим периодическую последовательность s^{20}, введенную ранее в примере 1.1.1. Ее профиль линейной сложности, имеющий вид:
1, 1, 1, 3, 3, 3, 3, 5, 5, 5, 6, 6, 6, 8, 8, 8, 9, 9, 10, 10,11, 11, 11, 11, 14, 14, 14, 14, 15, 15, 15, 17, 17, 17, 18, 18, 19, 19, 19, 19, ...,
изображен на Рис. 1.1. По графику видно, что линейная сложность не убывает. Кроме того, вертикальный скачок на графике может произойти только снизу от линии $L = N/2$. Если происходит скачок, то он симметричен относительно этой линии (см. [5]).

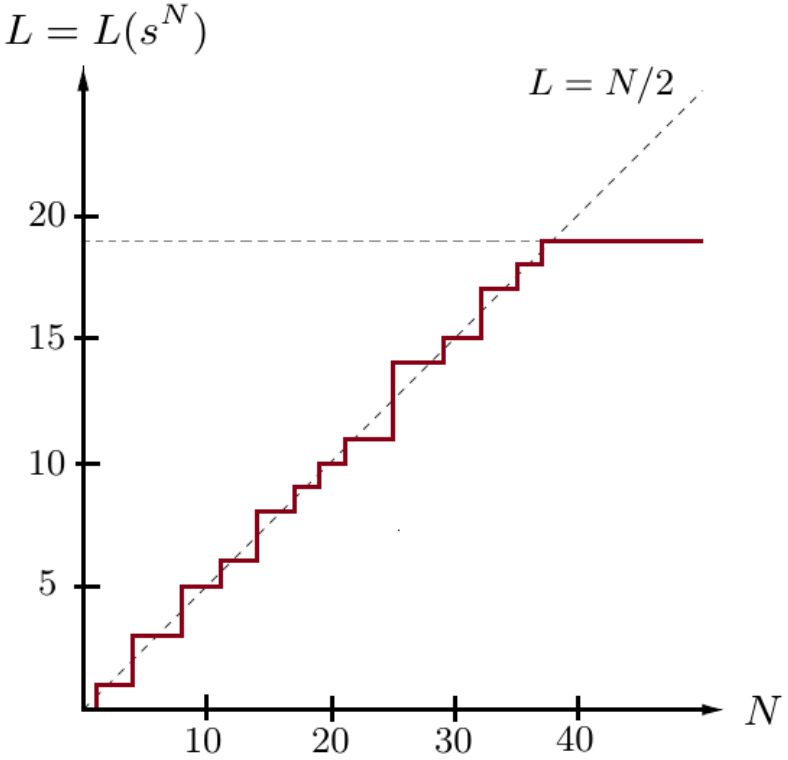

Рис. 1.1. Профиль линейной сложности 20-периодичной последовательности из примера 1.1.1

График отображает следующие свойства профиля (см. [5]):

- если $j > i$, то $L_j \geqslant L_i$;
- $L_{N+1} > L_N$ при $L_N \leqslant N/2$.

Rueppel, упомянутый ранее, также показал, что график профиля линейной сложности должен следовать линии $N/2$ близко, но не регулярно, как это видно на рисунке.

1.2. Обобщение линейной сложности

Понятие линейной сложности было обобщено до *k-error линейной сложности* — минимальной линейной сложности последовательности, в которой по крайней мере k элементов были изменены. Эта концепция была

впервые изложена Ding, Xiao, Shan [6], которые назвали ее *весовой сложностью*, и получила название k-error линейной сложности в работе авторов Stamp и Martin [2]. В отечественной литературе ее иногда называют *линейной сложностью последовательности с k-кратной ошибкой*. Это свойство последовательности является важным критерием безопасности потоковых шифров и применяется к проблеме идентификации криптостойких псевдослучайных последовательностей.

Величину, названную k-error линейной сложностью, можно интерпретировать как наихудшее измерение линейной сложности, при котором произошло k или менее ошибок. Данная работа посвящена нахождению k-error линейной сложности и изучению того, как изменение ее значения влияет на криптостойкость псевдослучайных последовательностей.

Глава 2

Задача нахождения k-error линейной сложности

2.1. Математическая постановка задачи

Задачу нахождения k-error линейной сложности конечной или периодической последовательности можно рассматривать как экстремальную задачу. Мы будем искать решение в пространстве векторов ошибок, а в качестве целевой функции возьмем k-error линейную сложность данной последовательности s. Ограничением будет служить количество ненулевых элементов в векторах ошибок, не превышающее значение k .

Пусть $s = s_0, s_1, s_2, \ldots$ — бесконечная последовательность периода N с элементами из поля K. Зафиксиксируем целочисленное значение k, $0 \leqslant k \leqslant w_H(s_0, \ldots, s_{N-1})$, где $w_H(s_0, \ldots, s_{N-1})$ — *вес Хэмминга* последовательности $s^N = (s_0, \ldots, s_{N-1})$.

Определение 2.1.1. *Линейная сложность с k-кратной ошибкой (или k-error линейная сложность) бесконечной периодической последовательности s определяется как*

$$L_k(s) = \min_e \{L(s+e) \,|\, 0 \leqslant w_H(e_0, \ldots, e_{N-1}) \leqslant k\}, \tag{2.1}$$

где e — периодическая последовательность из поля K с периодом N.

Дадим аналогичное определение для конечных последовательностей.

Пусть $s = s_0, s_1, \ldots, s_{t-1}$ — конечная последовательность длины t с элементами из поля K. Зафиксиксируем целочисленное значение k, $0 \leqslant k \leqslant w_H(s)$, где $w_H(s)$ — вес Хэмминга последовательности s.

Определение 2.1.2. *k-error линейная сложность конечной последовательности s определяется как*

$$L_k(s) = \min_e \{L(s+e) \mid w_H(e) \leqslant k\}, \tag{2.2}$$

где e — конечная последовательность длины t из поля K.

Последовательности e называют *последовательностью ошибок* или *вектором ошибок.*

Определение 2.1.3. *Последовательность $L_0(s), L_1(s), L_2(s), \ldots$ называется профилем k-error линейной сложности последовательности s.*

Замечание 2.1.1. *В этой работе в качестве поля K будет использоваться поле Галуа $GF(2)$ с операцией сложения по модулю 2, то есть его элементами будут двоичные последовательности длины t.*

Тогда вес Хэмминга некоторой последовательности s, $w_H(s)$, будет определяться как количество единиц в этой последовательности.

В дальнейшем нам понадобится следующее свойство k-error линейной сложности:

Свойство 2.1.1 (См. [7])**.** *Для последовательности s (конечной или бесконечной) с элементами из поля $GF(2)$ выполняется неравенство*

$$L_i(s) \geqslant L_j(s) \quad \forall i < j.$$

Таким образом, в качестве экстремальной задачи будет рассматриваться задача нахождения величины (2.1) или (2.2). То есть, при заданном k, $0 \leqslant k \leqslant w_H(s^N)$, решается *экстремальная задача*

$$L(s+e) \to \min_e, \tag{2.3}$$

$$0 \leqslant w_H(e_0, \ldots, e_{N-1}) \leqslant k. \tag{2.4}$$

Следует заметить, что 0-error линейная сложность любой последовательности совпадает со значением линейной сложности. Отметим также, что 1-error линейная сложность последовательности (1.3) — это нуль, так как изменение последнего бита этой последовательности с 1 на 0 породит нулевую последовательность, линейной сложностью которой по определению равна нулю.

Если k-error линейная сложность последовательности слишком низкая при малых значениях k (например, k составляет меньше 10% от длины последовательности), то последовательность можно будет легко восстановить, зная часть ее подряд идущих символов (см.[7]). Поэтому небезопасно использовать ее в качестве ключевого потока для потокового шифра.

2.2. Вычисление k-error линейной сложности

Точный алгоритм для вычисления k-error линейной сложности бинарных последовательностей с периодом 2^m, $m \geqslant 1$, был предложен авторами Stamp и Martin [2]. Этот эффективный алгоритм является расширением алгоритма Games и Chan [8], предназначенным для вычисления линейной сложности таких последовательностей.

Не существует общего алгоритма для вычисления профиля k-error линейной сложности произвольной последовательности над произвольным конечным полем, кроме полного перебора, поэтому возможным будет только нахождение приближенного решения.

Алгоритм Stamp и Martin вычисляет точную k-error линейную сложности последовательности периода 2^n $(n \geqslant 1)$ и в данной работе использован в качестве вспомогательного. Код алгоритма находится в приложении Б.

Глава 3

Генетический алгоритм для вычисления k-error линейной сложности последовательности

В работе моделируется генетический алгоритм вычисления приближенной k-error линейной сложности. Основное внимание уделяется выбору параметров (размер популяции, число поколений, техника отбора, специальные типы кроссовера, специальные типы мутации, вероятность мутации, вероятность кроссовера), некоторые из которых зависят от размера входной последовательности и количества ошибок k.

Основной идеей генетического алгоритма является организация «борьбы за существование» и «естественного отбора» среди имеющихся решений. Так как генетический алгоритм использует биологические аналогии, то применяющаяся терминология также напоминает биологическую.

На входе алгоритма имеется последовательность s и целое значение k. На выходе алгоритма получаем приближение k-error линейной сложности данной последовательности s. Сформулируем основные понятия генетического алгоритма в терминах рассматриваемой задачи.

3.1. Хромосомы

Хромосомой или *особью* назовем любой возможный вектор длины N $e = (e_0, e_1, \ldots, e_{N-1})$ с элементами из поля $GF(2)$ веса не более k, то есть

$$w_H(e) \leqslant k. \tag{3.1}$$

В задаче нахождения k-error линейной сложности он рассматривает-

ся как вектор ошибок (см. 2.1). Тогда *геном* будет являться каждый бит вектора ошибок, а *популяцией* — конечный набор векторов ошибок.

В главе 2 говорится, что пространством поиска задачи (2.3) является пространство векторов ошибок. Определим его как

$$E_k = \{e \mid w_H(e) \leqslant k\}. \tag{3.2}$$

Принцип естественного отбора заключается в том, что в конкурентной борьбе выживает наиболее приспособленный. В рассматриваемой задаче приспособленность особи определяется целевой функцией (п. 3.2): чем меньше значение целевой функции, тем более приспособленной является особь. То есть лучшими хромосомами являются векторы ошибок, которые создают меньшую линейную сложность входной последовательности s.

Начальная популяция генерируется случайным образом. Создается последовательность длины t веса k, к которой применяется функция случайной перестановки random.shuffle().

3.2. Целевая функция

Качество каждой особи оценивается через целевую функцию, называемую *фитнесс-функцией* или *функцией пригодности*. Наша цель состоит в том, чтобы найти элемент $e \in E_k$, минимизирующий линейную сложность суммы $s + e$.

Фитнесс-функцию определим как

$$f : E_k \to \mathbb{Z}, \text{ где } f(e) = L(s + e).$$

Данная функция является целевой функцией задачи (2.3). Мы используем алгоритм Берлекэмпа–Мэсси для вычисления фитнесс-функции для каждой особи из всей популяции.

В работе [7] отмечается, что по результатам экспериментов пространство поиска раздроблено и имеется много локальных минимумов и максимумов. Рассматриваемая задача является задачей дискретной оптимизации, поэтому поиск минимальных и максимальных значений достаточно сложен.

3.3. Генетические операторы

Необходимо задать операторы генетического алгоритма. К ним относятся отбор, кроссовер и мутация.

3.3.1. Отбор родителей

В генетическом алгоритме требуется на основе исходной популяции создать новую, так чтобы векторы ошибок в новой популяции были бы ближе к исходному глобальному минимуму целевой функции. Для этого сформируем из исходной популяции пары родителей для скрещивания и опишем этот процесс более подробно.

Берется популяция, объем которой обозначим через $NumOfIndividuum$. На каждой итерации выбирается одна пара родительских особей, затем к ним применяется кроссовер. Число пар родителей в нашем алгоритме равно размеру популяции. Родительские особи могут повторяться.

Рассмотрим два оператора выбора родителей.

- **Метод рулетки (Roulette Wheel Selection, $RWSEL$).**

  Особи отбираются с помощью $NumOfIndividuum$ запусков рулетки, где $NumOfIndividuum$ — размер популяции.

  Для каждой особи $e^{(i)}$ вычисляется ее значение целевой функции, $f\left(e^{(i)}\right)$ $(i = 0, 1, \ldots, NumOfIndividuum)$. Затем вычисляется значе-

ние «общей» целевой функции всей популяции:

$$TF = \sum_{i=0}^{NumOfIndividuum-1} \frac{1}{f\left(e^{(i)}\right)}.$$

Колесо рулетки содержит по одному сектору для каждого члена популяции. Размер i-го сектора пропорционален вероятности попадания в новую популяцию $P\left(e^{(i)}\right)$, вычисляемой по формуле:

$$P\left(e^{(i)}\right) = \frac{1/f\left(e^{(i)}\right)}{TF} = \frac{1}{f\left(e^{(i)}\right) \cdot TF}.$$

Замечание 3.3.1. *Задача нахождения k-error линейной сложности является задачей минимизации, поэтому вероятность выбора каждой особи обратно пропорциональна ее целевой функции.*

При таком отборе члены популяции с более высокой приспособленностью (то есть с меньшей целевой функцией) будут выбираться чаще, чем особи с низкой приспособленностью.

- **Отборочный турнир (Tournament Selection,** $TSEL$**).**

  Из текущей популяции, содержащей $NumOfIndividuum$ особей, выбираются случайные пары особей, затем в каждой паре выбирается лучшая особь и сохраняется в массив родителей, которые в дальнейшем будут скрещиваться для создания потомков.

  Следующие два шага повторяются $NumOfIndividuum$ раз:

  1. Случайным образом выбираются два значения i_1 и i_2, $0 \leqslant i_1 < i_2 \leqslant NumOfIndividuum - 1$.
  2. Если $f\left(e^{i_1}\right) > f\left(e^{i_2}\right)$, тогда выбирается e^{i_2}.
     Иначе выбирается e^{i_1}.

Преимуществом данного способа является то, что он не требует дополнительных вычислений.

В качестве основного оператора выбора родителей выбран метод рулетки, так как по результатам экспериментов (см. п. 4.2) с его помощью решение находится быстрее.

3.3.2. Кроссовер

Для удобства чтения в дальнейшем мы будем обозначать родительские хромосомы через $p^{(1)}$ и $p^{(2)}$ (в программе им будут соответствовать Parent1 и Parent2). Также через $s^{(1)}$ и $s^{(2)}$ обозначим двух потомков (Son1 и Son2 соответственно), получившихся при помощи кроссовера, примененного к $p^{(1)}$ и $p^{(2)}$.

Следует заметить, что стандартные типы кроссовера не подходят для нашей задачи, так как вес особей не должен превышать заданное значение k. Возьмем одноточечный кроссовер и покажем на конкретном примере, что данный оператор приведет к превышению количества единиц, нарушив условие (3.1), наложенное на вектора ошибок. Рассмотрим в качестве родителей два вектора ошибок длины $t = 8$ с количеством единиц $k = 3$:

$$p^{(1)} = (\underbrace{0,1,0,0}_{p_1^{(1)}}, \underbrace{1,1,0,0}_{p_2^{(1)}}), \tag{3.3}$$

$$p^{(2)} = (\underbrace{1,0,1,0}_{p_1^{(2)}}, \underbrace{0,1,0,0}_{p_2^{(2)}}) \tag{3.4}$$

и скрестим их следующим образом: в $s^{(1)}$ последовательно скопируем элементы $\left(p_1^{(1)}, p_2^{(2)}\right)$, тогда $s^{(2)}$, наоборот, получит набор $\left(p_1^{(2)}, p_2^{(1)}\right)$, в резуль-

тате потомки будут выглядеть следующим образом:

$$s^{(1)} = (0, 1, 0, 0, 0, 1, 0, 0),$$
$$s^{(2)} = (1, 1, 0, 0, 1, 0, 1, 0).$$

Вес $w_H(s^{(2)}) = 4$, что противоречит условию (3.1). Аналогично, двухточечный и равномерный случайный кроссоверы также не подходят для нашей задачи, поэтому введем некоторые модификации стандартных типов. Модифицируем их следующим образом.

При выборе двух родителей $p^{(1)}$ и $p^{(2)}$ будем рассматривать следующие схемы кроссовера.

- **Специальный тип одноточечного кроссовера (Special Single Point Crossover, $SSPX$).**

  Пусть $IndLen$ обозначает длину каждой особи.

  1. При помощи функции random.randint() случайным образом выберем число $i_1 \in [0, IndLen]$, выполняющее роль индекса-разделителя.
  2. Элементы потомка $s^{(1)}$, соответствующие индексам с 0-го по $i_3 - 1$ включительно, заполняем элементами из $p^{(1)}$, соответствующими тем же самым индексам. Аналогичным образом $s^{(2)}$ копирует часть родителя $p^{(2)}$.
  3. Места потомка $s^{(1)}$, соответствующие индексам с i_3 до IndLen включительно, заполняем элементами из $p^{(2)}$, соответствующими тем же самым индексам. Аналогичным образом $s^{(2)}$ копирует часть родителя $p^{(1)}$.
  4. Если в потомках количество единиц превышает значение k, то единицы, выбранные при помощи функции random.randint(), за-

меняются на нули до тех пор, пока потомки не будут удовлетворять ограничению задачи.

Пусть $i_1 = 3, k = 3$, тогда, применив данный тип кроссовера для родителей (предварительно поставив разделитель)

$$p^{(1)} = (0, 1, 0 \,|\, 0, 1, 1, 0, 0);$$
$$p^{(2)} = (1, 0, 1 \,|\, 0, 0, 1, 0, 0),$$

после шагов 1–3 получим следующих потомков:

$$s^{(1)} = (0, 1, 0 \,|\, 0, 0, 1, 0, 0);$$
$$s^{(2)} = (1, 0, 1 \,|\, 0, 1, 1, 0, 0).$$

Второй потомок не удовлетворяет ограничению задачи. Пусть выбрана единица, стоящая на месте $i_2 = 2$. Тогда после 4-го шага потомки будут иметь окончательный вид:

$$s^{(1)} = (0, 1, 0 \,|\, 0, 0, 1, 0, 0);$$
$$s^{(2)} = (1, 0, 0 \,|\, 0, 1, 1, 0, 0).$$

- **Специальный тип кроссовера, использующий целевую функцию (Special Crossover using Loss Function, $SXLF$).**

  Пусть f_{p^1} и f_{p^2} есть целевые функции родительских особей p^1 и p^2 соответственно, а s^{ch} — потомок, полученный в результате применения оператора кроссовера над p^1 и p^2.

  Обозначим через $p_0(i)$ частоту появления значения 0 и через $p_1(i)$ частоту появления 1 для i-го гена в популяции. Тогда для всех i, $1 \leqslant i < IndLen$, определим s_i^{ch} следующим образом:

  1. Если $p_i^1 = p_i^2$, присвоим $s_i^{ch} = p_i^1 = p_i^2$.

2. Если $p_i^1 \neq p_i^2$:

   (a) При $p_i^1 = 0$, присвоим $s_i^{ch} = p_i^1$, если $f_{p^1} \cdot p_0(i) \geqslant f_{p^2} \cdot p_1(i)$. Иначе присвоим $s_i^{ch} = p_i^2$.

   (b) При $p_i^1 = 1$, присвоим $s_i^{ch} = p_i^1$, если $f_{p^1} \cdot p_1(i) \geqslant f_{p^2} \cdot p_0(i)$. Иначе присвоим $s_i^{ch} = p_i^2$.

Пусть $k = 3$. Рассмотрим следующих родителей:

$$p^{(1)} = (0, 1, 0, 0, 1, 1, 0, 0);$$
$$p^{(2)} = (1, 0, 1, 0, 0, 1, 0, 0).$$

Их целевые функции $f_{p^1} = f_{p^2} = 8$. Пусть

$$p_0(0) = 0.55, \quad p_1(0) = 0.45;$$
$$p_0(1) = 0.38, \quad p_1(1) = 0.62;$$
$$p_0(2) = 0.22, \quad p_1(2) = 0.78;$$
$$p_0(3) = 0.85, \quad p_1(3) = 0.15;$$
$$p_0(4) = 0.34, \quad p_1(4) = 0.66;$$
$$p_0(5) = 0.18, \quad p_1(5) = 0.82;$$
$$p_0(6) = 0.67, \quad p_1(6) = 0.33;$$
$$p_0(7) = 0.70, \quad p_1(7) = 0.30.$$

Тогда после 1 шага потомок будет иметь следующий вид:

$$s^{(ch)} = (\cdot, \cdot, \cdot, 0, \cdot, 1, 0, 0).$$

Применив к нему 2 шаг, получим:

$$s^{(ch)} = (1, 0, 1, 0, 1, 1, 0, 0).$$

$w_H(s^{(ch)}) = 4$, что не удовлетворяет ограничению задачи. Пусть случайно выбрана единица, стоящая на месте $i = 5$, заменена на ноль. Тогда окончательно потомок будет иметь вид:

$$s^{(ch)} = (1, 0, 1, 0, 1, 0, 0, 0).$$

3.3.3. Мутация

Мутация является оператором, необходимым для «выбивания» популяции из локального экстремума. За счет того, что при мутации изменяется случайно выбранный ген в особи, исключается преждевременная сходимость.

Мутации могут проводиться не только по одной случайной точке. Можно выбирать для изменения несколько точек в особи, причем их число также может быть случайным. Используют и мутации с изменением сразу некоторой группы подряд идущих точек.

Рассмотрим два типа мутаций.

- **Простая случайная мутация (Simple Random Mutation, SRM).**

  Такая мутация меняет бит в потомке, выбранный случайным образом.

  1. При помощи функции random.randint() случайным образом выбирается значение $i_1 \in [0, IndLen]$.
  2. В особи к i_1-ому элементу прибавляется 1. Сложение происходит по модулю 2.
  3. Если в мутирующей особи количество единиц превысило значение k, то в ней случайным образом берется i_2-ой элемент со значением 1 (какая-нибудь из единиц должна стать нулем) и заменяется на 0.

- **Мутация с переменной частотой (Variable Frequency Mutation, VFM).**

  Как обозначалось ранее для i-го гена в популяции:

  $p_0(i)$ — частота появления значения 0;

  $p_1(i)$ — частота появления значения 1.

  Вычисляется энтропия, соответствующая каждому i-ому гену:

  $$H_i = -p_0(i)\log p_0(i) - p_1(i)\log p_1(i). \tag{3.5}$$

  Определяется вероятность мутации i-го гена следующим образом:

  $$p_{mut}(i) = \frac{\dfrac{1}{H_i}}{\sum\limits_{j=1}^{IndLen}\dfrac{1}{H_j}}. \tag{3.6}$$

  При помощи функции random.randint() случайным образом в потомке выбираются 0 и 1. Пусть i_1 и i_2 — индексы, соответствующие им. Считаются их вероятности мутации $p_{mut}(i_1)$ и $p_{mut}(i_2)$. Все остальные гены потомка проверяются следующим образом.

  1.1 Если выбран ген со значением 1 (пусть он имеет индекс i_3), то проверяется условие $p_1(i) > p_0(i)$. Если оно выполнено, то проверяется условие

  $$p_{mut}(i_3) > p_{mut}(i_1). \tag{3.7}$$

  1.2 Среди всех генов, удовлетворяющих условию (3.7) выбирается ген с наибольшей p_{mut}, и в нем единица заменяется на нуль.

2.1 Если выбран ген со значением 0 (пусть он имеет индекс i_4), то проверяется условие $p_0(i) > p_1(i)$. Если оно выполнено, то проверяется условие

$$p_{mut}(i_4) > p_{mut}(i_1). \tag{3.8}$$

2.2 Среди всех генов, удовлетворяющих условию (3.8) выбирается ген с наибольшей p_{mut}, и в нем нуль заменяется на единицу.

В данной работе мутация следует за кроссовером, то есть оба типа мутации применяются к потомкам, полученным при помощи кроссоверов. В главе 4 выбор пары кроссовер–мутация будет сделан путем вычислений.

Дополнительные обозначения

- $NumOfIndividuum$ — размер популяции;
- $NumOfGeneration$ — число поколений;
- $P(SSPX)$ — значение из промежутка [0,1], представляющее вероятность выбора первого кроссовера;
- $P(SXLF) = 1 - P(SSPX)$ — значение из промежутка [0,1], представляющее вероятность выбора второго кроссовера;
- P_M — значение из промежутка [0,1], представляющее вероятность мутации;
- LC — линейная сложность последовательностей;
- LC_k — k-error линейная сложность последовательностей с данным значением k;
- LCP_k — профиль k-error линейной сложности последовательностей.

Глава 4

Численные эксперименты и результаты

4.1. Тесты с последовательностями длины 2^n

Генетический алгоритм, описанный ранее, может обрабатывать последовательности произвольной длины и вычислять приближенное значение k-error линейной сложности этих последовательностей. Для тестирования в качестве длины последовательности мы будем брать значения, равные степени 2. Такая локализация нужна для того, чтобы можно было вычислить точные значения k-error линейной сложности при помощи алгоритма Stamp и Martin. В дальнейшем тестирование будет проходить на бесконечных периодических последовательностях с данным полным периодом.

Рассмотрим бинарные последовательности с периодом $n = 2^5 = 32$ и различные значения параметров $NumOfIndividuum$, $NumOfGeneration$, $P(SSPX)$, P_M.

Начиная с k=0, будем запускать алгоритм, увеличивая значение k, пока k-error линейная сложность не станет равна нулю. Таким образом, мы получим профиль k-error линейной сложности последовательности s.

Мы применяем алгоритм Stamp и Martin для выборки из 10 последовательностей периода 32 с линейной сложностью 32 (таблица 4.1), результаты которого представлены в таблице 4.2. Видно, что выполняется основное свойство k-error линейной сложности: при увеличении k линейная сложность последовательностей уменьшается.

Таблица 4.1. Тестовые последовательности

Последовательность
$s_1 = [0,0,0,1,1,1,1,1,1,1,0,0,0,0,0,1,1,0,1,1,1,1,1,1,1,0,1,0,0,0,0,0]$
$s_2 = [1,1,0,1,1,0,0,0,1,0,0,1,0,0,1,0,1,1,0,0,0,0,0,0,0,0,1,1,1,0,1,0]$
$s_3 = [0,1,0,0,1,1,0,1,1,0,0,1,1,1,0,0,0,0,0,0,1,1,0,0,1,1,0,1,1,1,0,0]$
$s_4 = [0,1,0,0,0,0,0,0,0,1,0,1,1,0,1,1,1,0,1,1,1,1,0,1,1,1,1,0,1,1,1,1]$
$s_5 = [0,0,1,0,1,0,1,0,1,1,0,0,0,0,1,1,1,0,0,1,1,1,0,1,1,1,0,1,1,1,0,0]$
$s_6 = [1,1,0,0,0,0,1,1,0,0,1,0,1,1,0,1,1,0,1,0,0,0,0,0,1,1,1,1,1,1,0,1]$
$s_7 = [1,0,1,1,0,1,1,1,0,1,1,1,0,0,0,0,0,1,0,0,0,1,1,1,0,1,1,0,1,0,0,1]$
$s_8 = [1,0,0,0,0,1,0,0,0,0,0,1,1,1,1,0,0,0,0,0,0,1,1,1,1,1,0,1,1,1,1,0,0]$
$s_9 = [0,0,1,0,0,1,0,0,0,1,1,1,0,0,0,1,1,0,1,0,1,1,0,0,0,0,0,1,0,1,1,0]$
$s_{10} = [0,0,0,1,0,1,0,0,0,0,1,0,1,1,0,0,1,1,0,1,1,1,1,1,1,0,0,0,1,0,0,0,0]$

Таблица 4.2. Профиль k-error линейной сложности бинарных последовательностей

s_i	Профиль k-error линейной сложности
s_1	32, 29, 29, 21, 21, 9, 9, 9, 9, 9, 9, 9, 9, 3, 3, 1, 1, 0
s_2	32, 29, 29, 20, 20, 13, 13, 10, 10, 5, 5, 4, 4, 0
s_3	32, 22, 22, 15, 15, 11, 11, 3, 3, 3, 3, 3, 2, 2, 0
s_4	32, 29, 29, 21, 21, 17, 17, 17, 17, 17, 17, 9, 9, 1, 1, 1, 1, 1, 1, 0
s_5	32, 26, 26, 23, 23, 17, 17, 17, 17, 17, 17, 9, 9, 3, 3, 1, 1, 0
s_6	32, 27, 27, 25, 25, 21, 21, 9, 9, 9, 9, 5, 5, 5, 5, 1, 1, 0
s_7	32, 25, 25, 25, 25, 21, 21, 17, 17, 7, 7, 5, 5, 2, 1, 1, 0
s_8	32, 25, 25, 25, 25, 25, 25, 9, 9, 7, 7, 5, 5, 0
s_9	32, 29, 25, 25, 25, 19, 19, 17, 17, 6, 6, 3, 3, 0
s_{10}	32, 29, 29, 25, 25, 17, 17, 17, 17, 17, 17, 5, 5, 2, 2, 0

4.2. Сравнение типов отбора родителей

В этом разделе будут представлены результаты сравнения турнирного отбора ($TSEL$) и метода рулетки ($RWSEL$) для выбора родителей.

Из таблицы 4.1 возьмем последовательность s_2 в качестве тестовой. Вычислим для нее весь профиль k-error линейной сложности, выбирая родителей сначала методом рулетки, затем — турнирным отбором. Критерием выбора оператора будут служить не только результаты численных экспериментов, но и время, затраченное на их получение.

Зафиксируем основные параметры алгоритма, операторы кроссовера и мутации. Выбранные значения показаны в таблице 4.3.

Таблица 4.3. Выбранные параметры и операторы

$NumOfGeneration$	$NumOfIndividuum$	Тип кроссовера	Тип мутации
2000	150	$SXLF$	VFM

Для последовательности s_2 программа запускалась 5 раз, фиксировалось время, требуемое для вычисления k-error линейной сложности для каждого значения k, выбирались наименьшие значения целевых функций, полученные разными типами отбора. Результаты вычисления представлены в таблице 4.4.

Таблица 4.4. Профиль k-error линейной сложности s_2

k	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Точное значение	32	29	29	20	20	13	13	10	10	5	5	4	4	0
$RWSEL$	32	29	29	20	20	13	15	10	10	5	7	5	4	0
$TSEL$	32	29	29	20	20	13	15	11	10	7	5	4	4	0

Через $t_{RWSEL}(k)$ обозначим время в минутах, затраченное на поиск k-error линейной сложности с помощью метода рулетки. Тогда t_{RWSEL} — суммарное время, требуемое для вычисление всего профиля с помощью метода рулетки. Аналогично обозначим времена $t_{TSEL}(k)$ и t_{TSEL} для турнирного отбора. Полученные результаты зафиксированы в таблице 4.5.

Таблица 4.5. Время, затраченное на поиск LCP_k последовательности s_2

k	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$t_{RWSEL}(k)$	0	0	0	0	6	12	56	42	37	33	19	17	43	58
$t_{TSEL}(k)$	0	0	0	1	11	16	63	43	41	30	21	16	67	79

Таким образом, суммарное время:

$$t_{RWSEL} = 323 \text{ мин.}, \tag{4.1}$$

$$t_{TSEL} = 388 \text{ мин.} \tag{4.2}$$

Видно, что результаты метода рулетки и турнирного отбора практически совпадают, однако второй оператор значительно уступает первому по времени и тем самым является менее эффективным.

4.3. Сравнение типов кроссовера и мутации

В данной работе рассматриваются специальный тип одноточечного кроссовера ($SSPX$) и специальный тип кроссовера, использующий целевую функцию ($SXLF$). Проведем ряд экспериментов, и определим, какой из них лучше подходит для нахождения минимума целевой функции (2.3).

Зафиксируем основные параметры алгоритма: порядок сложности k, число итераций $NumOfGeneration$, длину последовательности $IndLen$, размер популяции $NumOfIndividum$ (таблицы 4.6 и 4.7).

Таблица 4.6. Выбранные параметры алгоритма

k	$NumOfGeneration$	$IndLen$	$NumOfIndividuum$
4	1000	32	100

Таблица 4.7. Выбранные параметры алгоритма

k	$NumOfGeneration$	$IndLen$	$NumOfIndividuum$
9	2000	32	200

Протестируем оба типа кроссовера на рассмотренной ранее выборке особей (таблица 4.1) с каждой мутацией (SRM и VFS) по отдельности.

Для каждого вектора s_i, $i = 1, \ldots, 10$, программа запускалась 10 раз, затем выбиралось наименьшее значение целевой функции, которое выдавала программа.

Результаты, полученные с применением простой случайной мутации (SRM) представлены в таблицах 4.8 и 4.9.

Результаты, полученные с применением мутации с переменной частотой (VFS) представлены в таблицах 4.10 и 4.11.

Эксперименты показывают, что специальный тип кроссовера, использующий целевую функцию ($SXLF$), дает значения фитнесс-функций чаще всего меньшие, чем специальный тип одноточечного кроссовера ($SSPX$). Таким образом, $SXLF$ больше подходит для генетического алгоритма, так как нас интересует минимизация фитнесс-функции.

Поэтому распределим вероятности кроссоверов следующим образом:

$$P(SSPX) = 0.3, \quad P(SXLF) = 0.7.$$

Также стоит заметить, что в некоторых случаях пара ($SXLF$, VFM) дает несколько лучшее значение целевой функции, чем пара ($SXLF$, SRM).

Таблица 4.8. Сравнение кроссоверов, $k = 4$, SRM–мутация

s_i	$SSPX$	$SXLF$	LC_4
s_1	21	21	21
s_2	20	20	20
s_3	16	15	15
s_4	21	22	21
s_5	24	23	23
s_6	26	25	25
s_7	25	25	25
s_8	27	25	25
s_9	26	25	25
s_{10}	26	25	25

Таблица 4.9. Сравнение кроссоверов, $k = 9$, SRM–мутация

s_i	$SSPX$	$SXLF$	LC_9
s_1	10	10	9
s_2	9	5	5
s_3	6	4	3
s_4	17	17	17
s_5	17	17	17
s_6	11	9	9
s_7	9	8	7
s_8	7	7	7
s_9	8	7	6
s_{10}	18	17	17

Таблица 4.10. Сравнение кроссоверов, $k = 4$, VFM–мутация

s_i	$SSPX$	$SXLF$	LC_4
s_1	22	21	21
s_2	22	20	20
s_3	17	15	15
s_4	22	21	21
s_5	24	23	23
s_6	27	25	25
s_7	25	25	25
s_8	25	26	25
s_9	26	25	25
s_{10}	25	25	25

Таблица 4.11. Сравнение кроссоверов, $k = 9$, VFM–мутация

s_i	$SSPX$	$SXLF$	LC_9
s_1	11	11	9
s_2	5	5	5
s_3	5	3	3
s_4	17	17	17
s_5	18	17	17
s_6	10	9	9
s_7	8	7	7
s_8	8	8	7
s_9	7	7	6
s_{10}	17	17	17

Поэтому в дальнейшем SRM–мутация будет применяться к потомкам, полученным при помощи $SSPX$–кроссовера, а VFM–мутация — к потомкам, созданным $SXLF$–кроссовером.

Теперь определим, какая мутация дает лучшее значение целевой функции. На рассмотренной ранее выборке особей (таблица 4.1) будем тестировать мутации с вероятностью 0,3, примененные к потомкам, созданным $SXLF$–кроссовером.

Зафиксируем основные параметры алгоритма: порядок сложности k, число итераций $NumOfGeneration$, размер популяции $NumOfIndividum$, вероятность мутации P_M и тип кроссовера (таблица 4.12). Результаты вычислений представлены в таблицах 4.13 и 4.14.

Таблица 4.12. Выбранные параметры алгоритма

k	$NumOfGeneration$	$NumOfIndividuum$	P_M	Кроссовер
7, 9	2000	150	0,3	$SXLF$

Эксперименты показывают, что в некоторых случаях мутация с переменной частотой (VFM) дает значения фитнесс-функций меньшие, чем простая случайная мутация (SRM). Однако чаще всего результаты вычислений обоих типов мутаций совпадают. В дальнейших экспериментах будем использовать пары кроссовер–мутация следующим образом: ($SSPX$, SRM) и ($SXLF$, VFM).

4.4. Тестирование с различными параметрами алгоритма

Применим генетический алгоритм к последовательностям длины 28 с фиксированным весом, меняя параметры алгоритма. Следует заметить,

Таблица 4.13. Сравнение мутаций, $k = 7$, $SXLF$–кроссовер

s_i	SRM	VFM	LC_4
s_1	9	9	9
s_2	10	10	10
s_3	7	5	3
s_4	17	17	17
s_5	17	17	17
s_6	9	9	9
s_7	17	17	17
s_8	10	10	9
s_9	17	17	17
s_{10}	17	17	17

Таблица 4.14. Сравнение мутаций, $k = 9$, $SXLF$–кроссовер

s_i	SRM	VFM	LC_9
s_1	11	11	9
s_2	5	5	5
s_3	4	3	3
s_4	17	17	17
s_5	17	17	17
s_6	10	10	9
s_7	8	8	7
s_8	8	8	7
s_9	7	6	6
s_{10}	17	17	17

что точное значение k-error линейной сложности рассматриваемых последовательностей неизвестно, так как 28 не является степенью 2, и алгоритм Stamp и Martin не сможет его вычислить.

Экспериментальным путем выясним, какие параметры будут давать наименьшее значение целевой функции.

Таблица 4.15. Последовательности длины 28 веса 15

Последовательность
$s_1 = [1,0,0,1,0,0,1,0,1,0,1,1,0,0,0,0,1,0,1,1,1,0,1,1,1,0,1,1]$
$s_2 = [0,1,1,0,0,1,1,0,1,1,0,0,1,1,1,0,1,1,0,0,1,1,0,0,1,0,0,1]$
$s_3 = [1,0,1,0,1,0,1,0,1,1,1,1,0,1,1,0,0,1,0,0,0,0,0,1,1,0,1,1]$
$s_4 = [1,0,1,0,0,1,1,1,1,1,0,0,1,0,0,1,0,0,0,0,1,1,1,1,0,0,1,1]$
$s_5 = [0,0,0,0,1,0,1,1,0,1,0,1,0,0,1,1,1,1,1,1,1,0,0,1,1,1,0,1,0]$

4.4.1. Размер популяций и их количество

Рассмотрим разные значения параметров алгоритма: число популяций и размер каждой популяции, а именно (100, 50); (500, 100); (2000, 150).

При вычислении использованы $SXLF$–кроссовер и VFM–мутация с фиксированными вероятностями 0,7 и 0,3 соответственно.

Для каждого вектора s_i, $i = 1, \ldots, 5$ (таблица 4.15), программа запускалась 10 раз и запоминались те значения фитнесс-функций, которые среди этих итераций были наименьшими. Результаты вычисления представлены в таблицах 4.16–4.20.

Таблица 4.16. Различные параметры популяций, s_1

Размер, число	Профиль k-error линейной сложности
100,50	28, 22, 22, 9, 13, 11, 12, 9, 16, 10, 12, 10, 15, 10, 13, 9
500,100	28, 22, 22, 9, 9, 9, 9, 9, 10, 6, 10, 9, 13, 10, 13, 9
2000,150	28, 22, 22, 9, 9, 9, 9, 9, 7, 6, 10, 3, 5, 5, 7, 2

Таблица 4.17. Различные параметры популяций, s_2

Размер, число	Профиль k-error линейной сложности
100,50	28, 19, 19, 12, 16, 10, 11, 13, 15, 9, 12, 12, 13, 10, 13, 11
500,100	28, 19, 19, 12, 12, 9, 10, 10, 4, 9, 10, 10, 10, 9, 4, 4
2000,150	28, 19, 19, 12, 12, 9, 9, 9, 4, 6, 6, 5, 4, 1, 4, 1

При выбранном числе популяций, равным 100 и размере каждой популяции, 50, алгоритм не дает даже приближенного решения при $k \geqslant 4$. Практически на каждой итерации нарушается основное свойство k-error

Таблица 4.18. Различные параметры популяций, s_3

Размер, число	Профиль k-error линейной сложности
100,50	28, 22, 19, 15, 15, 13, 13, 9, 16, 13, 15, 13, 10, 7, 13, 12
500,100	28, 22, 19, 15 10, 9, 10, 8, 10, 6, 10, 8, 9, 9, 9, 9
2000,150	28, 22, 19, 10, 10, 9, 9, 6, 5, 6, 6, 5, 3, 3, 3, 0

Таблица 4.19. Различные параметры популяций, s_4

Размер, число	Профиль k-error линейной сложности
100,50	28, 21, 16, 16, 16, 7, 10, 11 15, 10, 13, 12, 12, 12, 13, 11
500,100	28, 21, 16, 16, 16, 7 10 7 10 9 10 6 10 9 10 9
2000,150	28, 21, 16, 16, 16, 7, 10, 3, 3, 3, 7, 5, 3, 3, 5, 1

Таблица 4.20. Различные параметры популяций, s_5

Размер, число	Профиль k-error линейной сложности
100,50	28, 22, 19, 9, 15, 9, 15, 9, 15, 13, 9, 10, 10, 10, 12, 12
500,100	28, 19, 9, 9, 9, 9, 12, 10, 10, 9, 7, 9, 6, 12, 9
2000,150	28, 19, 9, 9, 9, 9, 7, 7, 10, 7, 5, 7, 5, 3, 5, 1

линейной сложности:

$$L_i(s) \geqslant L_j(s) \quad \forall i < j, \tag{4.3}$$

где $L_k(s)$ — k-error линейная сложность последовательности s.

Пара (500,100) дает результаты лучшие по сравнению с (100,50), но также наблюдается явное отклонение приближенного решения от точного.

Видно, что пара (2000, 150) дает лучшее приближенное решение, чем (100,50), (500,100), но при некоторых k снова нарушается свойство (4.3),

поэтому для нахождения наилучшего результата этих значений недостаточно.

Однако, даже при таком количестве итераций алгоритму требуется много времени для того, чтобы сгенерировать заданное количество популяций. А именно, для нахождения полного профиля k-error линейно сложности одной последовательности длины 28 с размером популяции 150 и количеством популяций 2000 требуется около 5,5 часов. Само вычисление проводилось на компьютере с четырехъядерным процессором Intel Core i5.

Исходя из вышестоящего, будем считать пару (2000, 150) оптимальной для последовательностей длины 28.

4.4.2. Вероятность мутации

Из таблицы 4.15 возьмем последовательность s_2 в качестве тестовой. Вычислим ее 5-error линейную сложность, используя мутацию с переменной частотой (VFM), так как в пункте 4.3 мы показали, что она дает лучшее значение целевой функции по сравнению с простой случайной мутацией. Критерием выбора подходящей вероятности будут служить не только результаты численных экспериментов, но и время, затраченное на их получение.

Зафиксируем основные параметры алгоритма (таблица 4.21) и будем менять вероятность мутации.

Таблица 4.21. Выбранные параметры алгоритма

k	$NumOfGeneration$	$NumOfIndividuum$	Кроссовер	Мутация
4, 6	1500	100	$SXLF$	VFM

Обозначим через t_M время в минутах, затраченное на поиск k-error линейной сложности и протестируем алгоритм с выбранными параметрами и

вероятностями мутации, равными 0,1; 0,2; 0,3; 0,4. Результаты представлены в таблицах 4.22 и 4.23.

Таблица 4.22. Сравнение мутаций с разными вероятностями

	LC_4	t_M, мин
$P_M = 0,1$	12	17
$P_M = 0,2$	12	15
$P_M = 0,3$	12	8
$P_M = 0,4$	12	9

Таблица 4.23. Сравнение мутаций с разными вероятностями

	LC_6	t_M, мин
$P_M = 0,1$	10	24
$P_M = 0,2$	10	24
$P_M = 0,3$	9	23
$P_M = 0,4$	9	23

Для каждого значения вероятности программа запускалась 5 раз, фиксировалось наименьшее полученное значение k-error линейной сложности и наименьшее время, за которое находилось решение.

Видно, что с вероятностями мутации, равными 0,3 и 0,4, в одном из случаев ($k = 6$) программа выдала более точный результат. Кроме того, решение было найдено быстрее, чем при значениях вероятности, равных 0,1 и 0,2. Однако, в случае $k = 4$ при $P_M = 0,3$ результат был найден несколько быстрее, чем при $P_M = 0,4$. Таким образом, можно считать, что выбор вероятности мутации, равной 0,3, является оптимальным.

Заключение

В данной работе был предложен генетический алгоритм для вычисления k-error линейной сложности последовательности произвольной длины над конечным полем $GF(2)$. Кратко перечислим основные результаты.

Реализованы различные методы для каждого эволюционного оператора (отбор, кроссовер, мутация) и исследован наилучший выбор параметров для этой задачи.

В настоящей работе рассмотрены следующие типы операторов:

- отбор родителей:
  - турнир;
  - метод рулетки;
- кроссовер:
  - специальный тип одноточечного кроссовера;
  - специальный тип кроссовера, использующий целевую функцию;
- мутация:
  - простая случайная мутация;
  - мутация с переменной частотой.

Вычислительные эксперименты показали, что генетический алгоритм дает хорошие результаты для рассмотренной задачи. Например, для последовательностей длины 28 следующая схема представляется оптимальной:

- среднее число популяций, 2000;
- среднее количество особей в популяции, 150;
- отбор родителей методом рулетки;

– специальный тип кроссовера, использующий целевую функцию, с вероятностью 0,7;

– специальный тип одноточечного кроссовера с вероятностью 0,3;

– мутация с переменной частотой с вероятностью 0,3.

С предложенными параметрами алгоритм выдает приближенное значение k-error линейной сложности, которая в большинстве построенных примерах всего на 14% выше точного значения.

Все результаты получены с помощью программы, с которой можно ознакомиться по ссылке https://yadi.sk/d/pstEP6aB3JRUdY/2017.

В продолжение данной работы можно рассмотреть другие потенциально эффективные стратегии. Например, в работе [7] в качестве основного оператора отбора родителей выбран *элитарный отбор 25%-го уровня*. В нем 25% лучших особей текущей популяции автоматически переходят в следующую популяцию. В качестве кроссовера можно рассматривать, например, *специальный тип двухточечного кроссовера*, работающий по тому же принципу, что и кроссовер $SSPX$ (см. пункт 3.3.2).

Список литературы

1. Massey J.L., Serconek S. Linear Complexity of Periodic Sequences // Lecture Notes in Computer Science. New York: Springer, 1996. V. 1109, P. 358–371.
2. Stamp M., Clyde F.M. An Algorithm for the k-Error Linear Complexity of Binary Sequences with Period 2^n // IEEE Transactions on Information Theory. 1993. V. 39, № 4, P. 1398–1401.
3. Xiao G., Wei S., Lam K.Y., Imamura K. A fast algorithms for determining the linear complexity of a sequence with period 2^n over $GF(q)$// IEEE Transactions on Information Theory. 2000, V. 46, № 6, P. 2203-2206.
4. Rueppel R.A. Analysis and Design of Stream Ciphers. Berlin: Springer-Verlag. 1986.
5. Menezes P., van Oorshot P., Vanstone S. Stream Ciphers// Handbook of Applied Cryptography. CRC Press. 1996. P. 191-221.
6. Ding C., Xiao C., Shan W. The Stability Theory of Stream Ciphers // Lecture Notes in Computer Science. Heidelberg: Springer-Verlag. 1992.
7. Alecu A., Salagean A.M. A genetic algorithm for computing the k-error linear complexity of cryptographic sequences // IEEE Congress on Evolutionary Computation. 2007. P. 3569–3576.
8. Games R.A., Chan A.H. A Fast Algorithm for Determining the Complexity of a Binary Sequence with Period 2^n // IEEE Transactions on Information Theory. 1983. V. 29, № 1, P. 144–146.
9. Нгуен М.Х. Применение генетического алгоритма для задачи нахождения покрытия множества // Динамика неоднородных систем. М.:Изд-вл ЛКИ. 2008. Т. 33. Вып. 12. С. 206-219.

Приложение А

Алгоритм Берлекэмпа–Мэсси

Программа написана на языке Python 3.5.

Входные параметры: двоичный массив s.

Выходные параметры: линейная сложность s.

```
import numpy as np
import random
s = [...]
n = len(s)
c = [0 for i in range(n)]; c[0] = 1
b = [0 for i in range(n)]; b[0] = 1
L = 0; m = -1; N = 0;
while N < n:
    sum = 0
    for i in range(1, L + 1):
        sum = (sum + c[i] * s[N-i]) % 2
    d = (s[N] + sum) % 2
    if d == 1:
        t = [c[i] for i in range(n)]
        for j in range(n - N + m):
            c[N - m + j] = (c[N - m + j] + b[j]) % 2
        if L <= N / 2:
            L = N + 1 - L
            m = N
            b = [t[i] for i in range(n)]
    N += 1
```

Приложение Б

Алгоритм Stamp, Martin

Программа написана на языке Python 3.5.

Входные параметры: двоичный массив a, значение k.

Выходные параметры: k-error линейная сложность a.

```
import numpy as np
import random
a = [...]
k = ...
l = len(a)
c = 0;
cost = [1 for i in range(l)];
j = 0
while l > 1:
    j += 1
    l = int(l/2)
    T = 0
    L = [a[i] for i in range(l)]
    R = [a[i+l] for i in range(l)]
    b = [(L[i] + R[i]) % 2 for i in range(l)]
    for i in range(l):
        T = T + b[i] * min(cost[i],cost[i+l])
    a = [0 for i in range(l)]
    if T <= k:
        k = k - T
        for i in range(l):
```

```
            if b[i] == 1:
                if cost[i] <= cost[i+l]:
                    L[i] = R[i]
                    cost[i] = cost[i+l] - cost[i]
                else:
                    cost[i] = cost[i] - cost[i+l]
            else:
                cost[i] = cost[i] + cost[i+l]
        a = [L[i] for i in range(l)]
    else:
        c += l
        a = [b[i] for i in range(l)]
        cost = [min(cost[i],cost[i+l]) for i in range(l)]
if (a[0] == 1) & (cost[0] > k):
    c += 1
    print(c)
if c==0:
    print('zero')
```